

High Availability in the HP OpenCall SS7 Platform

Fault tolerance in computer systems is discussed and high availability is defined. The theory and operation of the active/standby HP OpenCall solution are presented. Switchover decision-making power is vested in a fault tolerance controller process on each machine.

by **Brian C. Wyld and Jean-Pierre Allegre**

Our lives are increasingly dependent on our technology. Some things are trivial, like being able to watch our favorite TV program, while some are much more important, like medical equipment. When you start to look at the difference technology makes to our lives, starting from the convenience, you begin to appreciate the problems we would have if it were to break down.

Some breakdowns are merely irritable. Not being able to phone to arrange a night out isn't going to kill anyone. But when you can't phone for help in an emergency, then a lack of the service we all take for granted is a lot more serious. Being able to ensure that the technology we use every day keeps working is therefore a part of the functionality, as much as providing the service in the first place.

Although it would be nice if things never broke down, we all know that this is impossible. Everything has a flaw—entropy always gets us in the end. The unsinkable ship sinks, the uninterruptible power supply gets cut, the unbreakable plate proves to be only too breakable (usually with the assistance of a child).

We all depend in one way or another on the continued functioning of our technology, so we all have some dependence on the tolerance of that technology to the faults that will inevitably strike it. When a computer malfunction can disrupt the lives of millions of people, fault tolerance is not just nice, but absolutely necessary.

Computer Fault Tolerance

Computer fault tolerance covers a range of functionality. The important aspects to consider are the speed of recovery in the presence of a fault, the perception of the users in case of a fault, and the consequences to the application of a fault. With these in mind, the following degrees of fault tolerance are often defined.

Reliable Service. The system is built to be as reliable as possible. No effort to provide tolerance of faults is made, but every part of the system is engineered to be as reliable as possible to avoid the possibility of a fault. This includes both the hardware, with either overengineered components or rigorous testing, and the software, with design methods that attempt to ensure bug-free code and user interfaces designed to prevent operator mistakes. Reliability rates of 99.99% can be achieved.

High Availability. The emphasis is on making the service available to the user as much of the time as possible. In the event of a fault, the user may notice some inconsistency or interruption of service, but will always be able to reconnect to the service and use it again either immediately or within a sharply bounded period of time. A reliability rate of 99.999% is the target.

Continuous Availability. This is the pinnacle of fault tolerance. When a fault occurs, the user notices no interruption or inconsistency in the service. The service is always there, never goes away, and never exhibits any behavior that leads the user to think that a fault might have happened. Needless to say, this level is both difficult and expensive to obtain. The reliability rate is 100%.

In general, a fault tolerant system will be highly available in most parts, with touches of continuous availability.

Achieving Fault Tolerance

Fault tolerance is usually achieved by using redundant components. Based on the assumption that any component, no matter how reliable, will eventually either fail or require planned maintenance, every component in the system that is vital is duplicated. This redundancy is designed so that the component can be removed with a minimal amount of disruption to the operation of the service. For instance, using mirrored disk drives allows a disk to fail or be disconnected without altering the availability of the data stored on the disk.

The way the redundancy is designed varies according to the paradigm used in the system. There are several ways to build in the redundancy, depending on the use made of the redundant components and how consistency is maintained between them.

Multiple Active. The redundant components may in fact be used simply to provide the service in a load-sharing way. In this case, data and functionality are provided identically by all the components. The load from the users of the service is spread across the components so that each handles a part of the load. In the event of a component failure, the load is taken up by the others, and their load increases correspondingly.

Active/Standby. In this paradigm, there exist one or more active components, which provide the service to the users, and in parallel, one or more standby components. These provide a shadow to the actives and in the event of an active failing, change state to become active and take over the job. Variations on the theme involve one-to-one active/standby pairs, N actives to one standby, and N actives to M standbys.

Coupling

How the service is affected by the failure is also important. The redundant component, whether it was also providing the service or not, can be loosely or tightly coupled to the other components.

When loosely coupled, the redundant component only has a view of the state of the active at certain times—at the end of a transaction, for instance. This has the effect that a failure of the component while processing a user's request will lose the context. Any finished work will be unaffected, but work in progress is lost and the user must restart. However, the effort required to keep the standby coupled is low.

A tightly coupled component will remain in step with the component processing the request, so that it can take over seamlessly in the event of the fault. The workload is much higher because many more messages must be exchanged, and the speed of the operation may be slower to ensure that at every stage the standby is in step with the active.

Of course, many shades and granularities of loose versus tight coupling are possible in a single system.

Traditionally, hardware fault tolerant systems have been exponents of the tight coupling paradigm: two or more processors executing exactly the same instructions in synchronization, with the outputs selected on either an active/standby basis or by a voting system. Software systems have leaned more towards the loose coupling method, at various levels of granularity. For instance, there are database transactional paradigms in which user database accesses are bundled into transactions, and only once a transaction is committed does that transaction become certain and unaffected in the event of a failure.

Predicting and Measuring Fault Tolerance

Various statistical methods exist to measure the fault tolerance of a system in a quantitative manner. These usually use the standard measures of system failure such as MTBF (mean time between failures) and MTTR (mean time to repair), and are combined to give a forecast of application downtime. However, the values of downtime produced by such methods can be inaccurate, and sometimes bear little resemblance to the true values.

The main reason for this is that failures may not be isolated and uncorrelated, and this is very difficult to take into account. Simply predicting from the MTBF and MTTR that the chance of a single failure bringing down the entire system is very small is not realistic when the single failure will often provoke subsequent related failures, often in the part of the system trying to recover from the fault. Most fault tolerant systems and related statistical analysis are based on an assumption of a single failure, and systems are built to avoid a single point of failure. In practice, and in the often inconvenient real world, failures can happen together, and can cause other failures in turn.

The other assumption that causes trouble is that of silent failure. It is often assumed that when a component fails, it does so silently, that is, in a failure mode that doesn't affect other components. For instance, having a dual LAN between several computers to avoid the LAN's being a single point of failure doesn't help when a crashed computer decides to send out nonsense on all of its LAN interfaces, effectively preventing use of any LAN.

Downtime Causes

Things that cause downtime on systems can be grouped into several main categories. First is the obvious computer hardware failure. This may be caused by a component's lifetime being exceeded, by a faulty component, or by an out-of-specification component. Often, hardware failures in one component can cause other components to fail. Many computer systems are not constructed to allow a single component to fail or to be replaced without affecting other components. For instance, a failed disk drive on a SCSI bus will force the entire system to be halted for its replacement even though only one component has failed.

This often implies that avoiding the single point of failure means adding more hardware than might seem reasonable—a second SCSI controller card and chain for instance, so that the backup disk drive can be on a separate SCSI bus. Reliable hardware, coupled with a system built to allow hot-swappable components can do a lot to eliminate this source of downtime.

The second obvious cause of downtime is software failures. No software will ever be entirely bug-free. Even formal methods, quality reviews, and all the rest of the trappings of computer science cannot keep those elusive problems from slipping in. The main problem with bugs is that the ones that escape to released systems are usually in code that has not been well-tested. This may often be the code designed to recover from failures, since this is difficult to cover fully with testing. Recovery may also often involve a higher load than normal as standby processes become active, load files, and so

on, and this can often expose lurking bugs. The net effect is that when your application crashes and burns, your standby application, ready and waiting to continue the service, promptly follows it down in flames instead.

Another not so obvious but very real source of downtime is operator intervention. Although in theory, operators of a system will always follow procedures and always read the manual, in practice they are prone to typing `rm *` (deleting all of the files on the disk), and pulling out the wrong power plug. Even when the mistake is not so obvious, mistakes such as badly configured systems, too much load on a critical system, or enabling unneeded tracing or statistics can bring the system down.

No amount of clever fault tolerant algorithms or mathematically proven designs will help here. However, a carefully planned system configuration, with working defaults and a user interface that is designed to help the user make the correct choices by presenting the correct information in a timely and obvious fashion, can go a long way towards avoiding these sorts of problems. This is, unfortunately, an often neglected part of the system design, with attendant problems. When building a system that end users will use as a platform, with programming interfaces, the importance of providing usable interfaces becomes even greater.

Finally, there are the disasters. The absolutely unforeseeable, one-in-a-million, couldn't happen in a lifetime chances. Who could predict that the earthmoving equipment would knock down the pylon supplying the mains electricity to the computer center, which would fall onto the telecommunications lines, blowing every piece of data communications equipment, after which it would careen into the hole where the water main was being repaired, breaking it and flooding the basement where the batteries for the UPS are installed, completely destroying them? "Impossible," you might say, but it has happened. In such cases, geographically separated sites can prove to be the only possible solution if a 100% available system is really required. This does rule out certain forms of fault tolerance—any form of dual-ported hardware, for instance, or lockstep processors—but is possible with software fault tolerance techniques.

Telecommunications Fault Tolerance

The requirements on a fault tolerant system vary with the application. In telecommunications, we see different requirements being demanded depending on the element being addressed. On the billing services side, the requirements are biased towards ensuring that no data loss occurs. Limited application downtime is acceptable but any billing data should be safe. This sort of system is similar to the requirements of any database-oriented application, and technologies such as mirrored disks and reliable systems are usually sufficient.

For operations services, which provide the management of the network, certain essential administration and management functions should always be available so that control over the network is always maintained. In the service provision environment for which the HP OpenCall SS7 platform is designed, the essential requirements are to avoid disruption to the network, to have a continuously available service, and to avoid disruption to calls in progress in the event of a fault.

To avoid disruption to the network, the SS7 protocol provision has to avoid ever being seen as down by the network. This essentially means that in the event of a fault, normal protocol processing must resume within six seconds. Any longer than this and the SS7 network will reconfigure to avoid the failed node. This reconfiguration process can be very load-intensive and can cause overloads in the network. This effect is to be avoided at all costs.

To provide continuous availability requires that the application that takes over service processing from a failed application must be at all times in the same state with respect to its processing. This is also required to ensure that current calls in progress are not disrupted. The state and data associated with each call must be replicated so that the user sees no interruption or anomaly in the service.

HP OpenCall Solution

To fulfill all of these requirements for a telecommunications services platform is not an easy task. We chose to implement a simple active/standby high availability fault tolerance model that is capable of providing most customer needs.

To achieve high availability, we need to replicate all the hardware components (see Fig. 1). We have defined a platform as being a set of two computers (usually HP 9000 Series 800) interconnected by a dual LAN, equipped with independent mirrored disks and sharing a set of SS7 signaling interface units via redundant SCSI chains (see [Article 7](#) for more details).

The highest constraint on the system is to be able to perform a switchover in less than six seconds. The SS7 protocol MTP Level 2 running on the signaling interface unit can tolerate a 6-s interval without traffic. If this limit is exceeded, the SS7 network detects this node as down, triggering a lot of alarms, and we've missed our high availability goal.

In a nutshell, the high availability mechanism works as follows. One system is the active system, handling the SS7 traffic and controlling all the signaling interface units. In case of a failure on the active side, the standby system gets control of the signaling interface units and becomes the new active. During the transition, the signaling interface units start buffering the data. When the buffers are full (which happens rapidly), the signaling interface units start sending MTP Level 2 messages to the other end to signal a transient outage. If this outage lasts more than 6 s, the SS7 network detects this node as down, so it is critical that in less than 6 s, a new active system take over. The failure detection time is the most crucial one. We need to detect failures in less than four seconds to be able to perform a safe switchover.

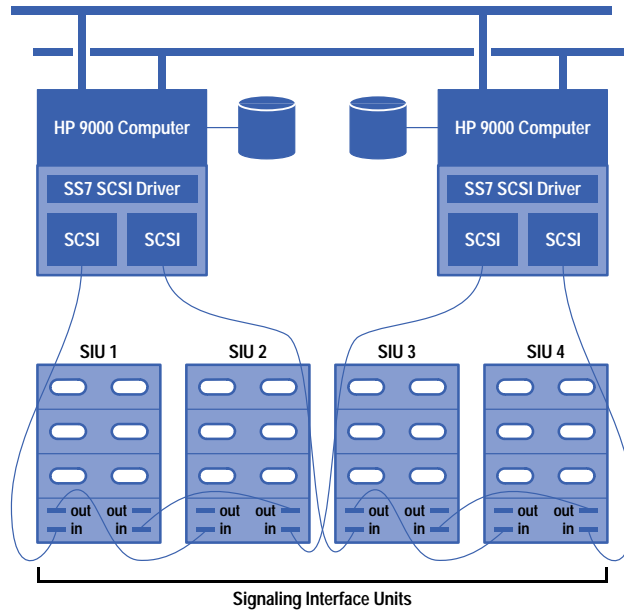


Fig. 1. The high availability solution in the HP OpenCall SS7 platform calls for replication of all hardware components.

The architecture goal is to protect the platform from failing in case of a single failure. In general, the dual-failure case leads to a service loss, even though some cases may be recovered.

Software Model

The HP OpenCall SS7 platform is based on an active/standby model, with a pair of UNIX[®] processes providing a service. Only one of the pair is actually doing the job at any time (the active) while its peer process (the standby) is idle, waiting to take over in the event of a failure.

The service provided by the process may be the SS7 protocol stack, centralized event management, or a telecommunications application. The standby process is not completely idle. It must be kept up to date with the state of the active process to be able to resume processing from the same point if a failure occurs. When in this state, it is *hot standby* (see Fig. 2).

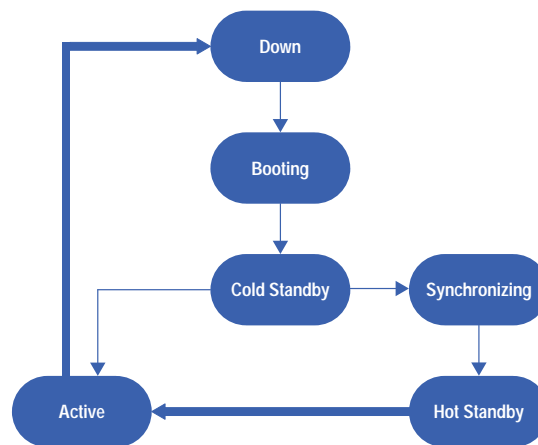


Fig. 2. A process starting up with an active process running is in the down state. When it reaches the hot standby state, it is ready to become the active if the active goes down.

Consider a process starting up when an active process is already running. A process is initially *down*, that is, not running. When it is started, it performs whatever startup process is required (*booting*), and then is *cold standby*. In this state, it is correctly configured and could perform the service if required, but all current clients would see their states being lost.

This would be enough to give a highly available service, but would not fulfill the requirement to avoid disruption to current clients. The process must therefore now synchronize itself with the active process, during which time it is *synchronizing*.

Once it is completely up to date, it is hot standby. In this state, current clients should see no disruption if the active process fails.

Obviously, if no active is running, the process goes to active from cold standby, since there are no current clients.

Once there exists this pair of processes, one active providing the service and one standby providing the backup, the system is ready to deal with a failure. When this occurs, the failure must first be detected, and then a decision on the action to be taken must be made.

Fault Tolerance Controller

The HP OpenCall SS7 platform centralizes the decision-making process into a single controller process per machine, which is responsible for knowing the states of all processes controlled by it on its machine. It has a peer controller (usually on the peer machine) which controls all the peer processes. These two *fault tolerance controllers* make all decisions with regard to which process of the pair is active. Each high availability process has a connection to both the fault tolerance controller and to its peer.

The fault tolerance controllers also have a connection between them (see Fig. 3). The A channel allows the two fault tolerance controllers to exchange state information on the two system processes and to build the global state of the platform. The A channel also conveys heartbeat messages. The B channels allow the fault tolerance controllers to pass commands governing the state of the processes, and to receive their state in return. A process cannot change state to become active without receiving a command from the fault tolerance controller. Because the fault tolerance controller has information on all high availability processes and on the state of the LAN, CPU, and all peer processes, it can make a much better decision than any individual process. Finally, the C channels are replication channels. They allow peer processes to replicate their state using an application dependent protocol.

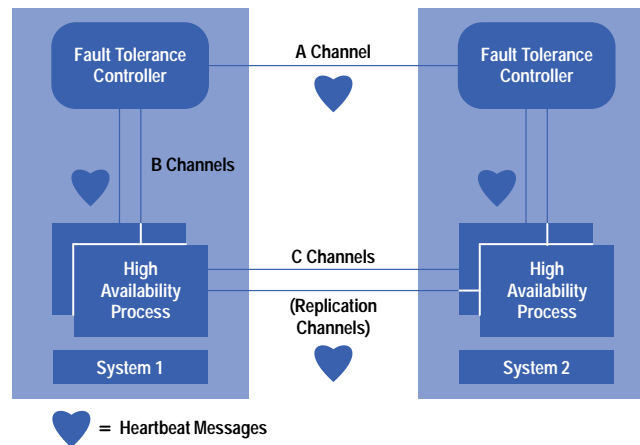


Fig. 3. The fault tolerance controllers make all decisions with regard to which process is active. Each high availability process has a connection to both the fault tolerance controller and to its peer. A heartbeat mechanism helps detect failures.

Failure Detection

For the success of any fault tolerant system, failures of any component must be detected quickly and reliably. This is one of the most difficult areas of the system. The HP OpenCall SS7 platform uses several mechanisms to detect various kinds of faults.

To detect a failure of one of the high availability processes, a heartbeat mechanism is used between the fault tolerance controller and the high availability process via the B channel. UNIX signals are also used to detect a failure of a child process (the fault tolerance controller is the parent of all high availability processes), but they provide information only when a process exits. To detect more subtle process faults such as deadlocks or infinite loops, the heartbeat mechanism is required, but it has the drawback of mandating that every high availability process be able to respond to heartbeat messages in a timely fashion, usually around every 500 ms. This is not so critical in our environment since we expect our processes to behave in a quasi-real-time manner, but it rules out using any potentially blocking calls.

To detect system hang, we use a specific mechanism implemented in the fault tolerance controllers. Each fault tolerance controller, running in HP-UX* real-time priority mode (rtprio), uses a real-time timer that ticks every 2 s (typically). At every tick, the fault tolerance controller checks the difference between the last tick and the current time. If this difference exceeds a certain tolerance, this means that the system has been hung for a while, since the fault tolerance controller is configured to have the highest priority in the system and should therefore never be prevented from receiving a real-time timer. Upon occurrence of such an event, the fault tolerance controller exits after killing the other high availability child processes. As

strange as it may sound, this is the safest thing to do. If the system has been hung for a while, the peer fault tolerance controller should have also detected a loss of heartbeat and should have decided to go active. If we were to let the fault tolerance controller of the hung system keep running once it wakes up, we would have two active systems, with all the possible nasty effects this entails.

The two fault tolerance controllers also exchange heartbeat messages (along with more detailed state information). Should a heartbeat fail, the fault tolerance controller of the active side will assume that the peer is down (for example, because of a dual-LAN failure, a peer system panic, or a peer fault tolerance controller failure) and will do nothing (except log an event to warn the operator). If the fault tolerance controller of the standby side detects this event, the fault tolerance controller will assume that something is wrong on the active side. It will decide to go active and will send an activate command to all of its high availability processes. If the old active is indeed dead, this is a wise decision and preserves the service. In the case of a dual-LAN failure (this is a dual-failure case that we are not supposed to guard against), we may have a split-brain syndrome. In our case, we use the signaling interface unit (see article, page 1) as a tiebreaker. If the active SS7 stack loses control of the signaling interface unit (because the peer stack has taken control of it), it will assume that the other system is alive and will exit, asking the fault tolerance controller not to respawn it. Operator intervention is necessary to clear the fault and bring the platform back into its duplex state.

Dual-LAN Support

At the time the HP OpenCall SS7 platform project started, no standard mechanism existed to handle dual LANs, nor did we want to implement a kernel-level dual-LAN mechanism. We therefore selected a user space mechanism provided by a library that hides the dual LAN and provides reliable message-based communication over two TCP connections (Fig. 4).

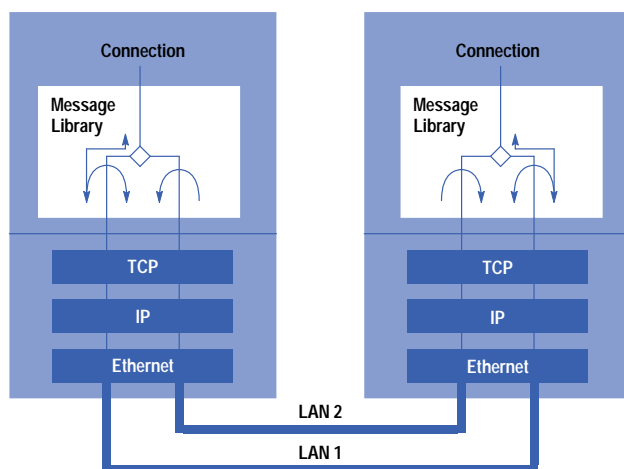


Fig. 4. The dual-LAN mechanism is a user space mechanism provided by a library that hides the dual LAN and provides reliable message-based communication over two TCP connections.

A *message library* provides the dual-LAN capability and message boundary preservation. The message library opens two TCP connections, one on each LAN. Only one LAN is used at a time (there is no attempt to perform load sharing). On each of the TCP connections, we maintain a small traffic of keep-alive messages (one every 500 ms), which contain just a sequence number. On each TCP connection, the message library monitors the difference between the sequence numbers on each LAN. If the difference exceeds a given threshold, one LAN is assumed to be either broken or overloaded, in which case the message library decides to switch and resume traffic on the other LAN. No heartbeat timer is used. Only differences in round-trip time can trigger a LAN switch. The benefit of this solution is that it is independent of the speed of the remote process or remote machine and scales without tuning from low-speed to high-speed LANs. It also allows very fast LAN switching time.

A drawback is the sensitivity of this mechanism to a loaded LAN, which is perceived as a broken LAN. For this, we recommend that an extra LAN be added to the system dedicated to application bulk traffic. Another problem is that when switching LANs, we have no way of retrieving unacknowledged TCP messages to retransmit on the new LAN, so we end up losing messages upon a LAN switch. Some parts of the platform guard themselves against this by implementing a lightweight retransmission protocol.

Access to High Availability Services

An important objective of the HP OpenCall SS7 platform is to shield the application writer from the underlying high availability mechanisms. We came up with the scheme illustrated in Fig. 5 to access the high availability processes.

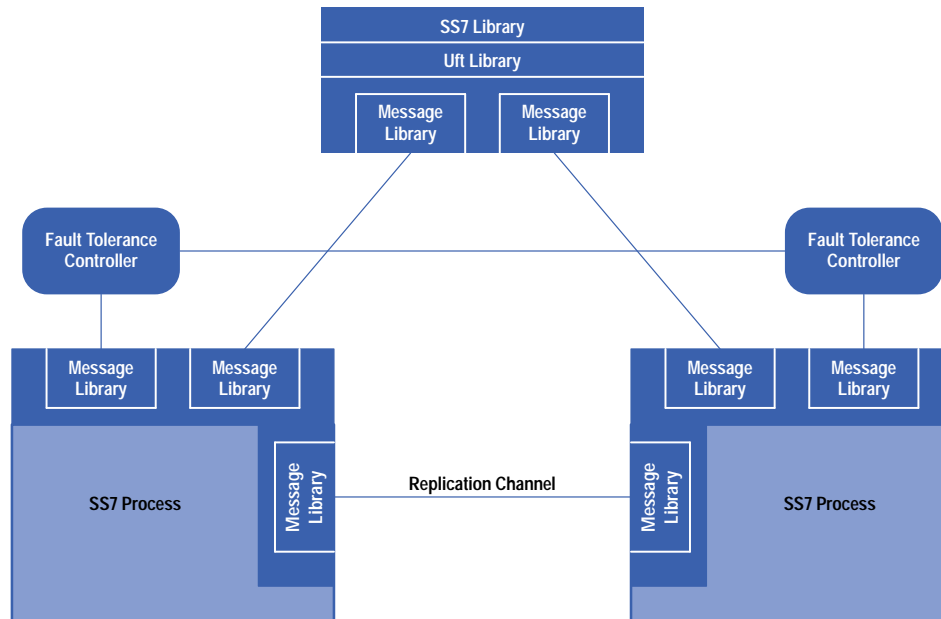


Fig. 5. The method of accessing the high availability processes is designed to shield the application writer from the underlying high availability mechanisms.

Let's take the example of the SS7 process. The *SS7 library* maintains two message library connections (four TCP connections because of the dual LAN): one with the active instance and one with the standby. The *Uft library* (user fault tolerance library) transparently manages the two connections and always routes the traffic to the active instance of the stack. Upon switchover, the SS7 process informs its client library via a surviving message library connection that it is now the new active and that traffic should be routed to it. From an API point of view, the two connections (four sockets) are hidden from the user by exporting an `fd_set` as used by `select()` instead of a file descriptor. The application main loop should be built along the following lines:

```
while(1) {
    API-pre-select(&rm,&wm,&em,&timeout);
    // application possibly adds
    // its own fd in the mask here
    // FD_SET(rm,myFd);
    select(&rm,&wm,&em,&timeout);
    API-post-select(&rm,&wm,&em,&timeout);
    // application possibly checks
    // its own fd here
    // if (FD_ISSET(rm,myFd)) {}
}
```

The application main loop must continuously call the preselect function of the API to get the accurate value of `fd_set` (sockets can be closed and reopened transparently in case of failure), then call `select()`, possibly after having set some application-specific file descriptor in the mask, then call the API postselect function with the mask returned by `select()`. In the postselect phase, the library handles all necessary protocol procedures to maintain an accurate view of the state of the high availability process, along with user data transfer.

State Management

One of the key aspects of the active/standby paradigm is the definition of the process state and how precisely it must be replicated. The framework described above does not enforce how state management should be performed. It provides a replication channel between the two peer processes and information about the processes, but no specific semantics for state. Different schemes are used depending on the nature of the state to be replicated. A key element to consider when designing such a system is the state information that must be preserved upon switchover and its update frequency. For instance, blindly replicating all state information in a system targeted at 4000 messages per second would be highly inefficient, because the replication load would exceed the actual processing.

For these reasons, we have not set up a generic state replication mechanism, but rather build ad hoc mechanisms depending on the nature of the state. For example, on the SS7 stack, the MTP 3 protocol has no state associated with data transfer (such as window value, pending timer, connection state), but has a lot of network management information that must not be lost in case of switchover.

The policy has been to intercept MTP 3 management messages coming from the network or from the OA&M (operation, administration, and maintenance) API and send them to the standby via the replication channel or the API. The standby stack processes the MTP 3 management messages in the same way as the active and the computed states are identical.

TCAP transactions are not replicated because of the high rates of creation and deletion and the amount of state information associated with the component handling. The effect is that opened TCAP transactions are lost in case of switchover. Work is progressing on a scheme that preserves transactions by letting the user of the transaction decide when the transaction becomes important and should be replicated.

An alternative to replicating messages is to replicate the state after it has been computed from the message. The usual algorithm for this scheme is to do the computation on the active side, use an ad hoc protocol to marshal the new state to the standby, and let the standby update itself. If the standby fails to replicate the state, it decides to go to the down state and will be restarted.

Another important design aspect for the high availability system is the synchronization phase. A starting cold standby system has to perform the cold standby to hot standby transition by getting all the state information from the active and rebuilding it locally. This operation should disturb the active as little as possible, but care must be taken that the algorithm converges. If the state of the active changes faster than the standby can absorb, there is a risk that the standby may never catch up. This is usually addressed by assuming that the standby has much more CPU available than the active, and if necessary, slowing down the active. In the case of SS7 signaling, the amount of state information is rather small and the information is stable, so we use a relatively simple algorithm. The synchronization is performed by having the standby fork a helper process that, via the SS7 OA&M API, dumps the content of the state information to disk and then replays it to the standby via the API. To check that the state is correct and that the standby can go to hot standby, the standby stack initiates an audit phase that checks that the two configurations are identical. If this is not the case, the process is resumed. Otherwise, the state of the standby goes to hot standby. This is a simple implementation, but has proven to be sufficient for SS7.

Technical Challenges

Developing a high availability platform on the HP-UX operating system has been a great challenge, but we've obtained a very stable and operational product, deployed in hundreds of sites worldwide.

One of the technical challenges was that HP-UX is not a real-time operating system and we need determinism to handle the high availability aspects, especially with the very small reaction time (<6 s) that we are allowed. We've addressed this by forbidding some operations (like file system access and potentially blocking system calls) in time-critical processes such as the SS7 stack, by slicing every long-lived operation into small operations, and by trying to stay below the saturation limit (where response time starts to increase rapidly). For example, we recommend keeping overall CPU utilization below 85% and staying far below the LAN maximum bandwidth.

Another challenge was time synchronization between the various hosts. We do not need time synchronization between the hosts for proper operation, but some of our customers request it. We've used the NTP package (Network Time Protocol), which has proved to work reasonably well except when the clock drift between the hosts was too large to be compensated smoothly and NTP decided to suddenly jump the system clock to catch up. This caused problems for synchronization of events, and also fired our failure detection mechanisms. We resolved these problems using external clocks and configuring NTP in a controlled manner to avoid such time jumps.

HP-UX 9.* and 10.0 for HP 9000 Series 700 and 800 computers are X/Open Company UNIX 93 branded products.

UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.

X/Open is a registered trademark and the X device is a trademark of X/Open Company Limited in the UK and other countries.

-
-
- ▶ [Go to Next Article](#)
 - ▶ [Go to Journal Home Page](#)